!!! ACHTUNG - evtl. veraltet - ACHTUNG !!!

Diese Seite wurde zuletzt am 9. Juli 2014 um 10:27 Uhr geändert.

Hilfe

http://www.linux-ag.de/linux/LHB/node26.html

oder

```
linux:~# man bash
```

Bash als "Editor"

```
<- bzw. ->
<Backspace>
<Entf>
<Pos1>
<Ende>
<Strg> + <- (bzw. <Alt>)
<Strg> + -> (bzw. <Alt>)
<Alt> + <D>
<Alt> + <Backspace>
<Strg> + <K>
<Strg> + <L>
<Strg> + <Space>
<Strg> + <W>
<Strg> + <Y>
<Strg> + <Shift> + <->
```

Ein Editor ist meist bequemer!

Editoren

```
linux:~# **aptitude install mc**
linux:~# **<span style="color:red;">mcedit</font> <Dateiname>**
```

kann alternativ auch nach starten von „mc“ mit der Funktionstaste F3 (Anzeigen) bzw. F4 (editieren) aufgerufen werden

```
linux:~# **aptitude install nano**
linux:~# **<span style="color:red;">nano</font> <Dateiname>**
```

(nano ist heutzutage meist schon installier!?)

```
linux:~# **aptitude install vim**
```

```
 linux:~# **<span style="color:red;">vi</font> <Dateiname>**
```

(zumindest der „einfache“ vi wird installiert sein, meist jedoch schon der „improved“ vi!?)
http://tnerual.eriogerg.free.fr/vimqrc-ge.pdf

für alle gilt: existiert die Datei <Dateiname> noch nicht, wird sie neu angelegt

Sonderzeichen

```
 <Space> (Worttrenner)
 * ? [[|]] (Jokerzeichen)
 { }     (Klammerersetzung)
 $       (Variablen- und Kommandoersetzung; alte Art: `...`)
 & | ;   (Befehlsverkettung)
 > <     (Ein- / Ausgabeumleitung)
 ( )     (Subshells)
 !       (Zugriff auf Historie)
 ~       (Abkürzung für das Heimatverzeichnis)
 #       (Kommentarzeichen)
 " ' \   (Quoting)
```

Aufgaben

Umleitungen

```
 0    (Standardeingabe)
 1    (Standardausgabe)
 2    (Standardfehlerausgabe)
 <    (Eingabe aus Datei lesen)
 >    (Ausgabe in Datei umleiten)
 >>   (Ausgabe an Datei anhängen)
 2>&1 (Ausgabe + Fehler umleiten)
 linux:~# **ping -c 1 localhorst <span style="color:red;">></font> /dev/null
<span style="color:red;">2>&1</font>**
   (ein Ping ohne Standard- und Fehlerausgabe)
```

Aufgaben

Geschweifte Klammern

... generieren Dateinamen durch Kombination aller Möglichkeiten

Beispiele:

```
 linux:~# **mkdir ablage_<span style="color:red;">{</font>1,2,3,4,5<span
```

```
style="color:red;">}</font>**
   (erstellt fünf Verzeichnisse "ablage_1", "ablage_2",...)
 linux:~# **cp /etc/passwd<span style="color:red;">{</font>,.orig<span
style="color:red;">}</font>**
   (kopiert "/etc/passwd" nach "/etc/passwd.orig")
 linux:~# **cp /etc/passwd<span style="color:red;">{</font>.orig,<span
style="color:red;">}</font>**
   (kopiert "/etc/passwd.orig" nach "/etc/passwd")
```

Aufgaben

Pipes

Linker Befehl gibt Daten auf Standardausgabe aus

Rechter Befehl liest Daten von Standardeingabe

Pipe verknüpft die beiden direkt

Beide Befehle laufen gleichzeitig Hand-in-Hand

Datenübergabe analog zu einer „Eimerkette“

Es werden keine temporären Dateien benötigt

Der Fehlerkanal wird nicht automatisch gepipet

Beispiel:

```
 linux:~# **echo "12345" <span style="color:red;">|</font> grep --color
"3"**
 12<span style="color:red;">3</font>45
```

Aufgaben

Kommandoersetzung

Zuerst werden Joker (* und ?) ersetzt

danach werden Kommandoerzetzungen durchgeführt

schließlich wird die sich so ergebene Befehlszeile ausgeführt

Beispiel:

```
 linux:~# **echo "heute ist <span style="color:red;">$(</font> date +%A
<span style="color:red;">)</font>."**
   //(alte Schreibweise vernachlässige ich einfach mal!)//
```

Aufgaben

Prozessersetzung

von Anfang bis Ende gleicher Stand der „Datei“

Vermeidung von temporären Dateien

Syntax:

```
 befehlA <span style="color:red;"><(</font> befehlB <span
style="color:red;">)</font>
 befehlA <span style="color:red;">>(</font> befehlB <span
style="color:red;">)</font>
```

Beispiele

```
 linux:~# **cat <span style="color:red;"><(</font> ls -la <span
style="color:red;">)</font>**
```

(virtuelle Datei mit der Ausgabe von „ls -la“)

```
 linux:~# **<span style="color:red;">>(</font> grep "tcp" <span
style="color:red;">)</font>**
```

(virtuelle Datei, Eingabe von…)

weitere Möglichkeiten, Prozesse zu kombinieren (teilweise schon behandelt):

```
 befehlA <span style="color:red;">|</font> befehlB
 befehlA <span style="color:red;">$(</font> befehlB <span
style="color:red;">)</font>
 befehlA <span style="color:red;">;</font> befehlB
 befehlA <span style="color:red;">&&</font> befehlB
 befehlA <span style="color:red;">||</font> befehlB
 befehlA <span style="color:red;">&</font> befehlB
 befehlA <span style="color:red;">>></font> dateiD
 befehlA <span style="color:red;"><<</font> dateiD
 befehlA <span style="color:red;">>>></font> dateiD
 befehlA <span style="color:red;">2></font> dateiD
 befehlA <span style="color:red;">2>&1</font> dateiD
```

Aufgaben

Kommandozeilenverarbeitung

Folgendermaßen arbeitet die Bash eine Kommandozeile ab:

1. geschweifte Klammern werden ersetzt
2. Ersetzung von ~ durch das Heimatverzeichnis
3. Ersetzung von $1, $2,... durch die Parameter
4. Kommandoersetzung
5. Rechenausdrücke
6. Zerlegung der Zeile in Worte
7. Ersetzung von Jokern
8. Funktion / Programm / Alias / Built-In
9. Ausführung

Rechnen

Rechnen mit ganzen Zahlen

$[1] *(alte Schreibweise vernachlässige ich hier auch einfach mal!)*

Innerhalb von $[2] ist kein $ vor Variablen nötig, aber auch nicht schädlich

um Überraschungen bedingt durch den Zeichensatz zu vermeiden, muss die Umgebungsvariable LANG vorher per „LANG=“ geleert werden... wenn das nicht von Erfolg gekrönt ist, muss selbiges evtl. noch für „LANGUAGE“ und/oder „LC_ALL“ gemacht werden

Beispiel:

```
linux:~# **LANG=; echo $(( 4 * 512 ))**
2048
```

mächtigere Alternative: bc

```
linux:~# **echo "4 * 512" | bc**
2048
```

Shellskript (Beispiel)

eine einfache Textdatei

eine ausführbare Datei (oder Aufruf per bash)

links von Kommentaren steht ein #

Befehle Zeile für Zeile (bzw. durch ; getrennt)

#!/bin/bash in der ersten Zeile sagt dem System, womit die Datei interpretiert werden soll

andere Dateien (z. B. Funktionsbibliotheken,...) können per „source“ oder „.“ an der entsprechenden Stelle ausgeführt werden

/root/anderesShellSkript.sh

```
#!/bin/bash
# /root/anderesShellSkript.sh
echo "vor dem ersten Befehl!"
```

/root/shellskript.sh

```
#!/bin/bash
# /root/shellskript.sh
# 1. Kommentar
. anderesShellSkript
echo "1. Befehl"
sleep 3 # 2. Befehl (2. Kommentar)
echo "3. Befehl"
```

Ausführung und Ausgabe:

```
linux:~# chmod u+x shellskript.sh
linux:~# ls -l shellskript.sh
-rwxr--r-- 1 root root 117 8. Aug 12:00 shellskript.sh
linux:~# **./shellskript.sh** (oder "bash shellskript.sh")
vor dem ersten Befehl!
1. Befehl
3. Befehl
```

Aufgaben

echo (Befehl)

-n kein Zeilenumbruch am Zeilenende

-e interpretiert Backslashes

- \n Zeilenumbruch (Linefeed)
- \r Wagenrücklauf (Carriage Return)
- \t Tabulator (nächste 8er-Spalte)
- \a Glocke, Piep
- \123 Asciizeichen (Oktal 123)

Beispiel:

```
linux:~# **echo <span style="color:red;">-e</font> "MfG\nPatrick"**
MfG
Patrick
```

Aufgaben

here-doc

Nach einem Befehl kommt « und die Endekennung Die Endekennung kann beliebig gewählt werden Die Befehlszeile kann weitergehen Der Befehl bekommt alle folgenden Zeilen als Eingabe ... bis eine Zeile nur aus der Endekennung besteht Danach geht das Shellskript normal weiter

Beispiel:

```
linux:~# **cat <span style="color:red;"><<EOF</font> > /tmp/here.test
> Das hier
> ist ein
> Hier-Dokument.
> <span style="color:red;">EOF</font>**
linux:~# **cat /tmp/here.test**
Das hier
ist ein
Hier-Dokument.
```

Aufgaben

(Shell)Variablen

Enthält der Inhalt der Variablen ein Sonderzeichen (s. o.), so muss der gesamte Inhalt durch „“ oder // umschlossen werden oder die einzelnen Sonderzeichen per \ „escaped“ werden!

Ausgabe aller Shellvariablen per „set“

„...“ ← Variablen werden ersetzt

'...' ← Variablen werden nicht ersetzt

Typendeklaration per „typeset“ (z. B. -a = Array)

$0 Name des Shellskriptes

$$ Prozess-ID der Shell selbst

$# Anzahl der Parameter

$? Exitcode des letzten Befehls

Beispiel:

```
linux:~# **VARIABLE="Patrick"**
linux:~# **echo $VARIABLE**
Patrick
linux:~# **false**
linux:~# **echo $?**
1
```

Parameter (oder Aufrufargumente)

$1, $2, $3,...

immer Prüfen!!! (sie kommen vom „bösen“ Benutzer)

Minimierung der Verarbeitung bei vollständiger Unterstützung der UNIX-Optionssyntax durch die Verwendung von „getopts“ (Beispiel: siehe „while“)

Nach Abarbeitung eines Parameters kann dieser per „shift“ aus der Liste entfernt werden, alle anderen Parameter rücken auf. (Problem: Parameter mit dazugehörigen „Werten“)

Aufgaben

getopts (Befehl)

Builtin-Kommando der bash

Verarbeitung von mehreren komplexen Parametern

1. Argument

- String
- vorangestellter Doppelpunkt schaltet die Ausgabe von getopts ab
- jeder Buchstabe entspricht einer gültigen Option
- ein auf den Buchstaben folgender Doppelpunkt besagt, dass die Option einen Wert verlangt

2. Argument

- Variable
- enthält die aktuelle Option aus den übergebenen Parametern

auf den Wert je Option kann per $OPTARG zugegriffen werden

Beispiel: siehe „while“

Umgebungsvariablen

Setzen per:

```
export $VAR (übernimmt die Shellvariable ins Environment)
export VAR="12345" (Wertzuweisung und Übernahme)
```

Jeder Prozess hat sie (nicht nur Shells)

Werden automatisch an Kindprozesse vererbt

Aber: Shellvariablen werden nicht vererbt

Shell: Environment (= Umgebung) als Shellvariablen „sichtbar“

löschen per „unset“

/root/mybashvar.sh:

```
#!/bin/bash
# /root/mybashvar.sh
echo $MYBASHVAR
```

Ausführung:

```
linux:~# **export MYBASHVAR="Hallo"**
linux:~# **env | grep MYBASHVAR**
MYBASHVAR=Hallo
linux:~# **./mybashvar.sh**
Hallo
linux:~# **unset MYBASHVAR**
linux:~# **./mybashvar.sh**
linux:~#
```

Variablenumformung

'#' schneidet vorne kurz ab

'##' schneidet vorne lang ab

% schneidet hinten kurz ab

schneidet hinten lang ab was abgeschnitten soll, kann ein einfacher Text oder ein Muster mit Jokern sein Unterschied # zu ## bzw. % zu nur, wenn das Muster Joker enthält

Beispiel:

```
linux:~# **TEST="/usr/bin/test.sh"**
linux:~# **echo ${TEST%/*}**
/usr/bin
linux:~# **echo ${TEST##*/}**
test.sh
```

Aufgaben

test (Befehl)

Syntax:

```
test AUSDRUCK
```

oder

```
[[|AUSDRUCK ]]
```

Beispiele für Ausdrücke:

```
test -e datei
[[|-d verzeichnis ]]
[[|-x datei ]]
[[|-n "zeichenkette" ]]
[[|"zeichenkette" = "zeichenkette" ]]
[[|zahl -lt zahl ]]
[[|zahl -ge zahl ]]
[[|! ... ]]
[[|test -a test ]]
[[|test -o test ]]
weitere siehe "man test"!
```

Aufgaben

if, elif, else und fi (Konstrukt)

Falls Bedingung erfüllt ist (if)...

- Programm wurde erfolgreich ausgeführt
- test-Befehl

... mache das Eine (then)...

... falls nicht, aber eine andere Bedingung erfüllt ist (elif), mache das, falls nicht...

... mache das Andere (else)

zum Schluss folgt ein fi!

/root/if_else.sh

```
#!/bin/bash
# /root/if_else.sh
//if ! echo "$1" | egrep "^[[+-]]?[0-9]+$" > /dev/null; then//
  //echo "Der erste Parameter ist keine gueltige Zahl!"//
  //exit 1;//
//fi//
**if test 0 -gt $1; then**
  echo "Der Parameter ist kleiner als Null."
**elif [[|0 -lt $1 ]]; then**
  echo "Der Parameter ist groesser als Null."
**else**
  echo "Der Parameter ist Null!"
```

```
**fi**
```

Ausführung und Ausgabe:

```
linux:~# **./if_else.sh <span style="color:red;">-10</font>**
Der Parameter ist kleiner als Null.
linux:~# **./if_else.sh <span style="color:red;">0</font>**
Der Parameter ist Null!
linux:~# **./if_else.sh <span style="color:red;">666</font>**
Der Parameter ist groesser als Null.
```

case (Konstrukt)

case ... in ... Alternative)... esac

Joker sind erlaubt

Balken erlaubt mehrere Alternativen

Bei Sonderzeichen in „Alternative“ das „escapen“ nicht vergessen

;; schließt den Fall ab (wenn vorher kein exit zum Abbruch geführt hat

Beispiel: siehe „while“

for... Wortliste... done (Konstrukt)

Einleitung mit „for“

Schleifenvariable kann frei benannt werden

Zugriff innerhalb der Schleife mittels $

Zugriff von außen auf die Schleifenvariable ist nicht ohne weiteres möglich

nach „in“ kommt Wortliste

- wort1 wort2 ... wortX

oder

- $@, read, Kommandoersetzung

„break“ bricht eine Schleife sofort ab

„continue“ beginnt sofort mit dem nächsten Durchlauf

Beispiel 1 (/root/for.sh):

```
#!/bin/bash
# /root/for.sh
LANG ======
ZAHLEN="1 2 3 4 5 6 7 8 9 10"
**for ZAHL in $ZAHLEN; do**
  echo $(( $ZAHL * $ZAHL ))
**done**
```

Ausführung und Ausgabe:

```
linux:~# **./for.sh**
1
4
9
16
25
36
49
64
81
100
```

Beispiel 2 (/root/for_schleife.sh):

```
#!/bin/bash
# /root/for_schleife.sh
for pid in $( pidof sshd ); do
  if [ "neupat75" = $( ps -Ao user,pid \
    | grep $pid \
    | awk '{print $1;}' ) ]; then
    continue
  fi
  kill -KILL $pid
done
```

Aufgaben

while... do... done (Konstukt)

Die while-Schleife läuft, solange die Bedingung wahr ist

Die until-Schleife wird solange ausgeführt, bis die Bedingung wahr ist

/root/while_getopts_case.sh

```
while getopts ":ab:c" opt; do
  case $opt in
    a  ) echo "verarbeite Option -a" ;;
    b  ) echo "verarbeite Option -b"
```

```
          echo "der Parameter der Option ist: $OPTARG"
          ;;
    c  ) echo "verarbeite Option -c" ;;
    \? ) echo "Verwendung: test.sh [[-a]][-b|barg][[-c]] Datei"
          exit 1
  esac
done
LANG=; shift $(( $OPTIND – 1 ))
echo "Die angegebene Datei heißt $1"
```

Aufgaben

read (Befehl)

Interaktives Erfragen von Benutzereingaben

Lesen von Zeilen und Worten aus Textdateien

read schlägt fehl, wenn keine Eingabe erfolgt (Abfrage mit „if“ möglich)

Syntax:

```
read GANZEZEILE
read VARA VARB VARC VARREST
```

Beispiel 1:

```
#!/bin/bash
# /root/read_user.sh
echo -n "Bitte Quellverzeichnis angeben: "
if read SOURCE; then
  echo "Quellverzeichnis: $SOURCE."
else echo "Kein Quellverzeichnis!"; exit 1
fi
echo -n "Bitte Zielverzeichnis angeben: "
if read TARGET; then
  echo "Zielverzeichnis: $TARGET."
else echo "Kein Zielverzeichnis!"; exit 2
fi
if mv "$SOURCE" "$TARGET"; then
  echo "Umbenennen war erfolgreich. :-)"
else echo "Umbenennen ist fehlgeschlagen. :-("; exit 3
fi
```

Beispiel 2:

```
#!/bin/bash
# /root/read_system.sh
while read MONAT TAG UHRZEIT MELDUNG; do
```

```
  echo "-------------------------"
  echo "Datum:   $TAG. $MONAT"
  echo "Uhrzeit: $UHRZEIT"
  echo "Meldung:"
  echo "$MELDUNG"
done < <( tail -n 5 /var/log/messages )
```

Blöcke (Konstrukt)

Mit { und } können Befehlsblöcke erstellt werden

Ein solcher Block wird wie ein Befehl behandelt

Er hat eine gemeinsame Ein-/Ausgabe

Beispiel:

```
#!/bin/bash
# /root/block.sh
{
  N=1000
  while [[|$N -lt 10000 ]]
  do
    echo $N
    LANG=; N=$(( $N + 1 ))
  done
} | grep 7.*7
```

Ausführung:

```
linux:~# ./block.sh | wc -l
495
```

function (Konstukt)

ist ein kleines „Unter-Shellskript“ im Shellskript

wird wie ein Programm über Namen aufgerufen

hat eigene Parameter $1, $2,...

muss vor dem ersten Aufruf definiert werden

hat auch Standardeingabe/-ausgabe

Beispiel:

```
#!/bin/bash
```

```
# /root/function.sh
function userlist {
  USERSHELL=$1
  cat /etc/passwd | grep $USERSHELL
}
userlist /bin/bash | sort
```

Ausführung:

```
linux:~# **/root/function.sh**
root:x:0:0:root:/root:/bin/bash
user:x:1000:1000:Patrick Neumann,,,:/home/user:/bin/bash
```

Signale

signalisiert einfaches Ereignis

sendet Kernel oder Prozess an einen anderen

Signalnr. 1-32: Art des Ereignisses

- 1 = **HUP** = manchmal: Konfiguration neu laden
- 2 = **INT** = schlägt Prozessbeendung vor (<Strg> + <c>)
- 15 = **TERM** = schlägt Prozessbeendung vor (kill)
- 9 = **KILL** = beendet Prozess gewaltsam
- 20 = TSTP = friert Prozess ein (<Strg> + <z>, abfangbar)
- 19 = STOP = friert Prozess ein (nicht abfangbar)
- 18 = CONT = setzt Prozess nach Signal 19 oder 20 fort

(mehr dazu: „kill -l“ bzw. „man 7 signal“)

nur root kann Signale an fremde Prozesse senden

Ereignis kann mit folgenden Befehlen gesendet werden:

```
kill [[|SIGNAL ]] PROZESSID
killall [[|SIGNAL ]] PROZESSNAME
```

Shell bekommt Signal z. B. von <Strg> + <c>

Abfangen mittels „trap BEFEHLE SIGNAL“

so können vor einer friedlichen Beendigung ggf. noch Daten auf die Festplatte geschrieben werden

Beispiel:

```
#!/bin/bash
# /root/trap.sh
function signal {
  echo -e "\nSignal $1 abgefangen..."
  echo "... beende jetzt Prozess $$."
  exit
}
trap "signal INT"  INT
while true ; do echo -n "." ; sleep 1 ; done
```

Shell 1 ausführen, danach Shell 2 ausführen und in der 1. Shell sollte dann etwas passieren.

Shell 1:

```
linux:~# **/root/trap**
.....
Signal INT abgefangen...
... beende jetzt Prozess 12345.
```

Shell 2:

```
linux:~# **kill -INT $( pgrep trap )**
```

reguläre Ausdrücke

einfache reguläre Ausdrücke: grep und sed

erweiterte reguläre Ausdrücke: egrep (= grep -E), sed (-r) und awk

Suchmuster sind ähnlich wie Joker

* und ? haben aber eine andere Wirkung als in der Shell!

Tipp: mit der Option –color färbt egrep die Fundstelle ein!

```
.      ein beliebiges Zeichen
*      vorangegangenes Zeichen beliebig oft
?      vorangegangenes Zeichen 0 oder 1 mal
+      vorangegangenes Zeichen mind. einmal
^      verneint Zeichenklasse oder das nachfolgende
           Suchmuster muss am Anfang vorkommen
$      das voranstehende Suchmuster muss am Ende
           vorkommen oder leitet ein Skalar ein
|      Alternative Ausdrücke
       maskieren des nachfolgenden (Sonder-)Zeichens
d      Ziffer (genau wie [[0-9]])
D      Nicht-Ziffer
w      Buchstabe, Ziffer oder Unterstrich (bedingt auch Umlaute)
W      Nicht-Buchstabe, Nicht-Ziffer und kein Unterstrich
```

```
 r      Steuerzeichen für Wagenrücklauf
 n      Steuerzeichen für Zeilenvorschub
 t      Steuerzeichen für Tabulator
 f      Steuerzeichen für Seitenvorschub
 s      Leerzeichen oder Steuerzeichen
 < bzw. > leere Zeichenkette am Wortanfang bzw. -ende
 S      Nicht-Leerzeichen oder Nicht-Steuerzeichen
 [[abc]]        a, b oder c
 [[^abc]]       ein Zeichen, das nicht a, b, oder c ist
 [[0-9]]        Ziffer
 [[a-zA-Z]] Klein- oder Großbuchstabe
 [[^äöüÄÖÜ]]    kein deutscher Umlaut
 <nowiki>[[:space:]]</nowiki>   Leerzeichen, Tabulator,...
 <nowiki>[[:alnum:]]</nowiki>   Zahl oder Buchstabe
 (...)      gruppiert einen Ausdruck
 (...|...)  erlaubt zwei Alternativen (mehrfach möglich)
 {8}        vorangegangene Gruppe genau 8 mal
 {2,4}      vorangegang. Gruppe 2, 3 oder 4 mal
 {0,3}      vorangegang. Gruppe bis zu 3 mal
 {5,}       vorangegang. Gruppe 5 mal oder öfter
            (die Syntax {,5} funktioniert nicht!)
```

Beispiel mit grep:

```
 linux:~# **echo "123 ist kleiner als 321" | egrep --color "[[0-9]]"**
 <span style="color:red;">123</font> ist kleiner als <span
style="color:red;">321</font>
```

Beispiel-Muster:

```
 M..t           Mist aber nicht Meist
 M.*t           Mt, Mut, Mist, Meist
 Mau?l          Mal, Maul aber nicht Mauul
 Ja+!           Ja!, Jaa! aber nicht J!
 [^<nowiki>[[:space:]]</nowiki>]*   beliebig viele Nicht-Leerzeichen
 [[0-9]]+           mindestens eine Ziffer
 (ha)+          ha, haha, hahaha,...
 Ba(ch|um)      Bach oder Baum
 [[0-9]]{4}     genau vier Ziffern
 ([[0-9]]{2}){1,2}  zwei oder vier Ziffern
 ^$         Leerzeile
 ^<nowiki>[[:space:]]</nowiki>*$        Zeile mit nichts als Leerzeichen
 ^#         Zeile, die mit # anfängt (Kommentar)
```

grep (Befehl)

...

sed (Befehl)

macht Änderungen in Textströmen und arbeitet zeilenbasiert

wichtigste Operation: Suchen und Ersetzen

zusätzlich: Selektion von Zeilen ähnlich wie grep, head und tail

liest aus Dateien oder von Standardeingabe (0)

schreibt immer nach Standardausgabe (1)

arbeitet mit regulären Ausdrücken

```
Linuxprogramm | sed 'Befehl'
sed 'Befehl' Datei
sed -r 'Befehl' Datei
sed -e 'Befehl' -e 'Befehl' Datei
sed -f DateiMitBefehlen Datei
   ---> #!/bin/sed -f
sed -n (Ausgabe abschalten)
sed 'Befehl' Datei1 > Datei2
   (Umleitung nach Datei1 geht nicht!)
   ---> mv Datei2 Datei1
d      lösche zeilen (delete)
p      ausgeben (print)
s/.../.../ ersetzen Treffer einmal (substitute)
s/.../.../g    ersetze alle Treffer
s/...//g   lösche Treffer
s/.../...&.../ hänge (vorne und hinten) an Treffer an
c Newline  ersetze Zeile (change)
a Newline  hänge Zeile an (append)
i Newline  füge Zeile ein (insert)
r      Datei integrieren (read)
```

Beispiele:

```
linux:~# sed '1 d' datei
linux:~# sed '100,$ d' datei
linux:~# sed '/info/ d' datei
linux:~# sed 's/ROOT/root/' datei
linux:~# sed 's/patrick/Patrick/g' datei
linux:~# sed 's/ [[A-z0-9.]][A-z0-9.]*evil.com//g' access.log
linux:~# sed '/meyer/ c\
       --- Ersatzzeile ---'
linux:~# sed '/neumann/ r datei'
```

awk (Befehl)

Arbeitet sich Zeile für Zeile durch eine Textdatei

kennt Variablen, Zahlen, kann rechnen (auch Fließkomma)

gut zum quantitativen Auswerten von Logdateien

gibt standardmäßig den Text nicht aus

das neue nawk sollte dem alten awk vorgezogen werden!

```
Linuxprogramm | awk '{Befehl}'
awk '{Befehl}' Datei
awk -f DateiMitBefehlen Datei
   ---> #!/bin/awk -f
-F
{ Befehl; }
BEGIN { Befehl; }
END { Befehl; }
print „Text“
printf „Format“,Variablen
$1, $2,... $NF
+ - * /
n = n + 1
```

Beispiele:

```
linux:~# ls -l | awk '{print $2;}'
linux:~# ls -l | awk '{print $( NF-1 );}'
linux:~# awk -F : '{print $1;}' /etc/passwd
linux:~# awk -F : '{printf "%-8s %4d %-10s\n",$1,$2,$3;}' /etc/passwd
linux:~# awk -F : 'BEGIN{print "---"}
       {printf "%-8s %4d %-10s\n",$1,$2,$3;} END{print "---"}' /etc/passwd
linux:~# awk 'BEGIN{FS=":"} {print $1;}' /etc/passwd
linux:~# awk '/root/ {print $0}' /etc/passwd
linux:~# awk 'NR > 100 && NR < 200 {print $3}' access.log
linux:~# awk '/info/ {i = i + 1} END{print "anzahl: ",i}' error.log
linux:~# awk '/info/ {i++} END{print "anzahl: ",i}' error.log
linux:~# awk '/info/ {i = i + $3}
       END{print "ergebnis: ",i," (wie toll!)"}' datei
```

Verarbeitung von (mehrdimensionalen)Arrays

Rechnen

Verzweigungen (analog zu if)

Vergleichsoperatoren (Zahlen und Strings)

Logische Operatoren (&&, || und !)

Schleife (for und while inkl. Sprungbefehle)

Eingabe-Funktionen (analog zu read)

Systemaufrufe

Funktionen (analog zu function)

[1)]

RECHENAUSDRUCK

[2)]

…

From:
https://wiki.neumannsland.de/ - **Patricks DokuWiki**

Permanent link:
https://wiki.neumannsland.de/mw2dw:ds3000-bash

Last update: **2019/09/23 11:07**